Abgabe Übungsblatt 04

Technische Informatik 2 Wintersemester 2025/26

Baris Basar, Cengizhan Evcil, Nicolai Held

Aufgabe 1

Scheduling

Gegeben seien fünf immer lauffähige Prozesse mit den folgenden Eigenschaften:

Prozessname	Startzeitpunkt	Laufzeit	Basispriorität
P0	0	350	10
P1	20	50	0
P2	40	500	20
P3	110	450	60
P4	130	300	50

Ermittelt für die erstend acht Scheduling-Vorgänge, welcher Prozess jeweils gewählt wird und tragt ihre Name
sind die Zeitscheiben jeweils 50 ms lang. Die Laufzeit ist in Millisekunden angegeben. Bei gleichen Werten wir

Abarbeitungsreihenfolge der Prozesse bei Shortest Remaining Job First:

P0,P1,P1,P0,P0,P0,P4,P3

Abarbeitungsreihenfolge der Prozesse beim Kontenmodell:

P0,P1,P2,P0,P2,P0,P4,P2

Absenden

Flag: flag{ca7f3ad0-4a97-4ff3-9850-f22532a674f7}

Figure 1: Das ist die Flagge von “Scheduling”

a)

Um bei a) die Lösung zu ermitteln muss man sich immer genau anschauen welcher Prozess die kürzeste Laufzeit hat, die Priorität lässt sich nur davon ableiten[3]: Zunächst startet bei 0 der Prozess P0 (Laufzeit 350), daher wird P0 als erstees eingetragen. Dann, bei 20ms, startet der Prozess P1 mit seiner Laufzeit von 50ms (RestlaufZeit P0 bei 330ms). Da P1 nur 30ms Restlaufzeit hat, bei dem Zeitpunkt 40ms (Startpunkt von P2), läuft P1 weiter. Bei 70ms

ist der Prozess P1 abgeschlossen und P0 ist der nächste laufende Prozess, da er mit 330ms Restlaufzeit die 500ms von P2 unterbietet. Bei 110ms kommt der Prozess P3 mit einer Laufzeit von 450ms hinzu, allerdings ist der Prozess P0 bei einer Restlaufzeit von 290ms und läuft daher weiter. Bei 130ms kommt P4 hinzu (300ms), doch P0 ist der Shortest-Remaining-Job mit einer Laufzeit von 270ms. Wenn bei 400ms der Prozess P0 beendet wurde, dann wird erst P4 und danach endlich P3 abgearbeitet, da P4 kürzer ist mit 300ms und schließlich die 450ms von P3 abgearbeitet werden. So entsteht die Lösung aus dem Screenshot.

b)

Hier muss man am besten eine Tabelle aufstellen, um so die Werte vergleichen zu können. Je niedriger der Wert der Prio, desto besser. Der Job mit dem niedrigsten Wert wird also ausgeführt, nachdem ein Job für 50ms ausgeführt wurde, erhöht sich der Wert der Priorität zwangsweise. Der erste Wert in der Tabelle wird alle 50ms halbiert und setzt dann addiert mit der Basispriorität den zweiten Wert zusammen (Priorität). Das Ergebnis schaut also so aus:

	0	50	100	150	200	250	300	350
P0	0 / 10	50 / 60	25 / 35	12 / 22	56 / 66	28 / 38	64 / 74	32 / 42
P1	–	0 / 0	–	–	–	–	–	–
P2	–	0 / 20	0 / 20	50 / 70	25 / 45	62 / 82	31 / 51	15 / 35
P3	–	–	–	0 / 60	0 / 60	0 / 60	0 / 60	0 / 60
P4	–	–	–	0 / 50	0 / 50	0 / 50	0 / 50	50 / 100

Bei 0 wird also P0 ausgewählt, dieser Prozess hat bei 50ms noch 300ms Restlaufzeit. Bei 50ms wird P1 ausgewählt, dieser Prozess hat bei 100ms abgeschlossen. Bei 100ms wird P2 ausgewählt, dieser Prozess hat bei 150ms noch 450ms Restlaufzeit. Bei 150ms wird P0 ausgewählt, dieser Prozesss hat bei 200ms noch 250ms Restlaufzeit. Bei 200ms wird P2 ausgewählt, dieser Prozess hat bei 250ms noch 400ms Restlaufzeit. Bei 250ms wird P0 ausgewählt, dieser Prozess hat bei 300ms noch 200ms Restlaufzeit. Bei 300ms wird P4 ausgewählt, dieser Prozess hat bei 350ms noch 250ms Restlaufzeit. Bei 350ms wird P2 ausgewählt, dieser Prozess hat bei 400ms noch 350ms Restlaufzeit.

Alle Prozesse werden sinnvoll abgearbeitet. Der kürzeste Prozess P1 endet zb. nur 30ms später als beim SRTF, falls dieser Prozess allerdings keinerlei solcher Verzögerungen wünscht, dann ist der Prozess nicht sinnvoll abgearbeitet worden, da er sich um 30ms verzögert hat. Auch der Prozess P0 braucht eher doppelt so lang um abgearbeitet zu werden, allerdings wird P2 viel früher beendet als bei SRTF, was anhand dessen, dass die Priorität für P2 bei 20 liegt, ein enormer Vorteil gegenüber SRTF ist. Dort wird P2 als letztes abgearbeiet.

Aufgabe 2

2a. Einleitung

Ziel dieser Aufgabe ist es, eine einfache Shell zu implementieren, die:

- Kommandos mit beliebig vielen Argumenten einlesen kann,
- diese Kommandos als neue Prozesse startet,
- Kommandos entweder im Vordergrund oder im Hintergrund ausführt,
- typische Fehlerfälle berücksichtigt,
- Signale korrekt verarbeitet (SIGCHLD).

Ein Teil des Codes wurde vorgegeben (Einlesen der Befehle, Signalhandler, Shell-Struktur). Die Aufgabe bestand darin, die tatsächliche Prozessausführung mittels `fork()` und `execv()` einzubauen.

2b. Aufbau & Funktionsweise der Shell

Die Shell läuft in einer Endlosschleife:

```
auto [path, cmd] = Shell::readCommand();
```

`readCommand()` liefert:

- **path** → vollständiger Pfad zum auszuführenden Programm
- **cmd** → Objekt mit:
- **argv** → Argumentliste (vector)
- **background** → true, wenn Prozess im Hintergrund laufen soll (&)

Wird kein gültiges Kommando eingegeben, springt die Shell zum nächsten Durchlauf.

2c. Prozessverwaltung

Dies geschieht durch:

```
pid_t pid = fork();
```

Es gibt drei mögliche Fälle:

Rückgabewert	Bedeutung
-1	Fehler beim Erzeugen des Prozesses
0	Wir befinden uns im Kindprozess
>0	Wir befinden uns im Elternprozess, Wert = PID des Kindes

2d. Ausführen des Kommandos im Kindprozess

Der Kindprozess muss ein anderes Programm ersetzen. Dazu:

1. `vector<string>` wird in `char*[]` übersetzt → nötig für `execv()`
2. `execv(path, args)` wird aufgerufen

Bei Erfolg kehrt execv() niemals zurück.

2e. Verhalten des Elternprozesses

Vordergrundprozess → Shell blockiert, bis dieser Prozess beendet ist

Hintergrundprozess → Shell blockiert NICHT, Nutzer kann weiter tippen

Die Hintergrundprozesse werden vom vorhandenen `SignalHandler` über **SIGCHLD** eingefangen und aufgeräumt.

2f. Fehlerbehandlung

Die Shell behandelt:

Fehler beim `Fork`:

```
if (pid == -1) {
    perror("fork");
    continue;
}
```

Fehler beim `Exec`:

Falls `execv()` fehlschlägt:

- Falscher Programmpfad
- Keine Rechte
- Binärdatei beschädigt
- ...

Dann:

```
perror("execv");
exit(EXIT_FAILURE);
```

2g. Testfälle & Testergebnisse

1. `ls -l`

```
command: ls
arguments: -l
background: nein
```

2. `ls -l &`

```
command: ls
arguments: -l
background: ja
```

3. `sleep 10`

```
command: sleep
arguments: 10
background: nein
```

4. `sleep 10 &`

```
command: sleep
arguments: 10
background: ja
```

5. `echo foo`

```
command: echo
arguments: foo
background: nein
```

6. `echo foo &`

```
command: echo
arguments: foo
background: ja
```

7. `bla/fasel`

```
command: bla/fasel
arguments:
background: nein
bla/fasel: command not found
```

8. `sleep 100`

```
command: sleep
arguments: 100
background: nein
```

9. `mkdir test`

9.1. Erster Aufruf:

```
command: mkdir
arguments: test
background: nein
```

9.2. Zweiter Aufruf:

```
command: mkdir
arguments: test
```

```
background: nein
mkdir: test: File exists
```

10. `cd abc`

```
command: cd
arguments: abc
background: nein
No such file or directory
```

11. `date`

```
command: date
arguments:
background: nein
Thu Nov 20 21:37:47 CET 2025
```

12. `date &`

```
command: date
arguments:
background: ja
Thu Nov 20 21:37:50 CET 2025
```

13. `cat a.txt`

```
command: cat
arguments: a.txt
background: nein
cat: a.txt: No such file or directory
```

14. `cat a.txt &`

```
command: cat
arguments: a.txt
background: ja
cat: a.txt: No such file or directory
```

Literaturverzeichnis

1. https://man7.org/linux/man-pages/
2. https://cplusplusreference.com
3. https://www.studytonight.com/operating-system/shortest-remaining-time-first-scheduling-algorithm?utm_source=chatgpt.com (18.11.2025)

Anhang